PoV-Ray ismeretek (7. rész)

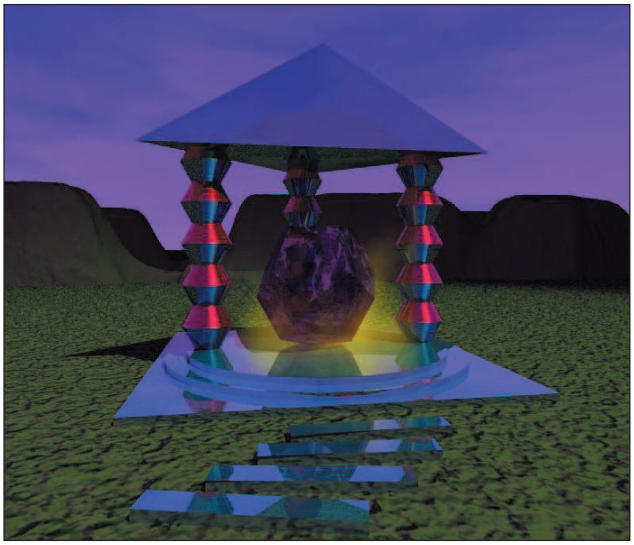
Bevezetés az animációkészítés rejtelmeibe

Mindeddig sikeresen elsajátítottuk azokat az ismereteket, amelyek elegendőek ahhoz, hogy szép állóképeket állítsunk elő, így jogosan merülhet fel a kérdés, hogy a PoV-Ray vajon alkalmas-e animációk készítésére. A válasz természetesen igen, és a cikksorozat utolsó két részében ezeket a lehetőségeket szeretném bemutatni.
A PoV-Rayben nem használhatunk úgynevezett kulcskocka-alapú animációt, ahol az egyes mozgásfázisokat, változásokat csak néhány képkocka számára kell meghatározni és a köztes képeken az egyes tárgyak, fények, kamerák helyzetét a program számolja ki. A PoV-Ray esetében minden képkockán meg kell adni minden tárgy tulajdonságát, és a program ezekből az adatokból állítja elő a képeket. Egy-egy tárgy mozgását függvényekkel írhatjuk le, ezért bemutatom azokat a függvényeket, amelyeket alkalmazhatunk és használatuk módját is igyekszem érthetően elmagyarázni.
A PoV-Ray programozói szerencsére számos függvényt beépítettek a programba, amelyekkel lebegőpontos számokon, vektorokon és szövegállandókon egyaránt műveleteket végezhetünk. Egy-egy függvényt a következő módon hívhatunk meg:

```
f ggvØnynØv(paramØter1,
paramØter2, ..., paramØterN)
```

Ez a hívási mód azonos a C és Pascal nyelvben megszokottakkal. Kezdjük az ismerkedést a lebegőpontos számokkal kapcsolatos függvényekkel, amelyek értékei lebegőpontos számok és eredményül szintén lebegőpontos számot kapunk.

- `abs(A)`: az értékként kapott `A` szám abszolút értékét adja vissza.
- `acos(A)`: Arcus Cosinus, eredménye az a szög radiánban mérve, melynek koszinusza az `A` érték.
- `asin(A)`: Arcus Sinus, eredménye az a radiánban mért szög, amelynek szinusza `A`.
- `atan2(A, B)`: az (`A`/`B`) érték arcus-tangensét kapjuk eredményül. Természetesen `B` értéke nem lehet 0. Egyetlen érték arcus-tangensének kiszámítására is használható a következő formában: `atan2(A,1)`. Erre akkor lehet szükségünk, ha a kiszámítandó szög tangense már ismert. A visszaadott értéket szintén radiánban szolgáltatja.
- `ceil(A)`: meghatározza a legkisebb egész számot, amely nagyobb `A`-nál, vagyis az értékként kapott számot felfelé kerekíti.
- `cos(A)`: a radiánban mért `A` szög koszinuszát adja vissza.
- `degrees(A)`: visszaadja az `A`-nak megfelelő szög fokban mért értékét a `deg=(180/Pi)*A` képlet felhasználásával.
- `exp(A)`: a természetes szám `A` kitevőjű hatványát adja vissza.
- `floor(A)`: az `A` számnál kisebb legközelebbi egész számot határozza meg.
- `int(A)`: a kerekítés matematikai szabálya szerint az `A` számot egészre kerekíti; ha `A` törtrészének értéke nagyobb 0,49-nél, akkor felfelé kerekít, ellenkező esetben lefelé.
- `log(A)`: az `A` szám természetes alapú logaritmusát határozza meg.
- `max(A, B)`: `A` és `B` számok közül a nagyobb értéket adja vissza.
- `min(A, B)`: `A` és `B` számok közül a kisebb értéket kapjuk eredményül.
- `mod(A, B)`: a maradékos osztás által szolgáltatott maradék meghatározása a `mod=A-floor(A/B)` képlet alapján.
- `pow(A, B)`: az `A` szám `B` kitevőjű hatványát határozza meg.



- `radians(A)`: a fokban mért `A` szög radiánértékét kapjuk eredményül.
- `rand(A)`: a következő álvéletlen számot kapjuk vissza – mielőtt használnánk ezt a függvényt, a véletlenszám-generátort a `seed()` függvénnyel be kell töltenünk; a függvény visszatérési értéke 0 és 1 közé esik.
- `seed(A)`: a véletlenszám-generátort alaphelyzetbe állítja – több véletlenszám-generátort is használhatunk, ekkor azonban célszerű különböző kezdőértékeket beállítanunk a számukra; az alábbi példa erre mutat egy lehetőséget:

```
#declare R1=seed(0)
#declare R2=seed(87568)

sphere {
    <rand(R1), rand(R1), rand(R1)>,
    rand(R2)
}
```

Láthatjuk, hogy egy változónak úgy adhatunk kezdőértéket – ezzel egyben a PoV-Rayjel is tudatjuk, hogy az adott azonosítót változóként értelmezze –, hogy a `#declare` kulcsszó után megadjuk a változó nevét és kezdeti értékét.

1. lista

```
#if (FELT TEL)
            utas tÆsok, ha a feltØtel igaz
      ...
#else
      utas tÆsok, ha a feltØtel hamis
#end
```

2. lista

```
#switch (AZONOSITO)
      #case (FELT TEL_1)
        Utas tÆsok, ha AZONOSITO
        ØrtØke = FELT TEL_1
      #break

      #case (FELT TEL_2)
        Utas tÆsok, ha AZONOSITO
        ØrtØke = FELT TEL_2
      #break

      #range (MIN_1, MAX_1)
        Utas tÆsok, ha MIN_1 <= AZONOSITO
        ØrtØke <= MAX_1
      #break

      #range (MIN_2, MAX_2)
        Utas tÆsok, ha MIN_2 <= AZONOSITO
        ØrtØke <= MAX_2
      #break
      #else
        Utas tÆsok, ha az elızı feltØtelek
        egyike sem teljes lt.
#end
```

- `sin(A)`: az `A` szög szinuszát szolgáltatja – természetesen a szöget itt is radiánban adjuk meg.
- `sqrt(A)`: az `A` szám négyzetgyökét adja vissza.
- `tan(A)`: ahogy már megszokhattuk, az `A` szöget szintén radiánban adjuk meg, e függvény eredménye pedig a szög tangense lesz.

Ezekkel a műveletekkel nagyon sok számítást elvégezhetünk a lebegőpontos számok körében, ezért ideje rátérni a vektorokkal kapcsolatos függvényekre.
Az alábbiakban bemutatott műveletek értékül egy vagy több vektort kapnak, némelyik egy lebegőpontos számot is, eredményük pedig vektor vagy lebegőpontos szám lesz.
Itt vektorként csak háromelemű vektort használhatunk, de a térbeli ábrázolás során valószínűleg ennél többre nincs is szükség. A jelölésekkel kapcsolatban még annyit említenék meg, hogy az `A` és `B` betűk vektort jelölnek, míg az `F` betű lebegőpontos számot.

- `vaxis_rotate(A, B, F)`: az `A`-t elforgatja a `B` vektor által meghatározott tengely körül `F` szöggel; a szöget fokban kell megadni és visszatérési értéke az elforgatott vektor.
- `vcross(A, B)`: `A` és `B` keresztszorzatát számítja ki; a visszaadott vektor merőleges `A`-ra és `B`-re, hosszúsága pedig arányos a közöttük lévő szöggel.
- `vdot(A, B)`: a két vektor skaláris szorzatát határozza meg a következő képlet alapján: `vdot=AxBx+AyBy+AzBz`.
- `vlength(A)`: a vektor hosszúságát adja meg – kiválóan használható két pont távolságának meghatározására, a felhasznált képlet a következő: `vlength= (vdot(A,A))`.
- `vnormalize(A)`: visszatérési értéke egy egységnyi hosszúságú vektor, melynek iránya megegyezik az értékként kapott `A` vektor irányával – a következő képlet használatával állítható elő: `vnormalize=A/vlength(A)`.
- `vrotate(A,B)`: eredménye az az elforgatott `A` vektor, amelyet az `X` tengely körül `Bx` szöggel – az `Y` tengely körül `By` szöggel –, a `Z` tengely körül pedig `Bz` szöggel forgatunk el; az elforgatás mértékét fokban adjuk meg.

A PoV-Rayben használható függvények utolsó csoportját alkotják azok, amelyek segítségével szövegeken végezhetünk műveleteket. Értékük egy vagy több szöveges állandó és esetlegesen lebegőpontos szám, visszatérési értékük pedig szöveg vagy szám lehet. Az alábbiakban `S1` és `S2` jelöli a szöveges értékét, míg az `A`, `L` és `P` jelek lebegőpontos számot vagy ilyen típusú eredményt szolgáltató kifejezést jelölnek. Természetesen egy-egy szám vagy vektor az előbbi függvények esetében is helyettesíthető a szükséges típusú eredményt adó kifejezéssel, függvénnyel.

- `asc(S1)`: visszaadja a szöveg első karakterének ASCII-kódját, például az asc(ABC) értéke 65 lesz, mert az A karakter kódja 65.
- `chr(A)`: azt a karaktert adja visszatérési értékül, melynek ASCII-kódja `A` – a kérdéses számnak 0 és 255 közé kell esnie, például a `chr(70)` értéke `F` lesz, mert ennek a karakternek a kódja 70; amikor ezt a függvényt szövegek megjelenítésére használjuk, figyelembe kell venni azt a tényt, hogy a 127-nél nagyobb kódok által meghatározott karakterek az adott karakterkészlettől függenek; a legtöbb TrueType font az ISO-8859-1-es (Latin-1, nyugat-európai) karakterkészletet használja, melyben a hosszú ő és a hosszú ű nem a magyar írásban használatos ékezetekkel jelenik meg.
- `concat(S1, S2 [,S3, ...])`: a függvény használata során legalább két karakterláncot kell megadnunk, eredményül pedig egy szöveges állandót kapunk, amely az értékekként megadottakat összefűzve tartalmazza; az alábbi példa jól szemlélteti a függvény használatát:

```
#declare cat_str=concat("Alma", "fa")
```

  ennek eredménye az „Almafa” szó lesz.
- `file_exists(S1)`: amennyiben az `S1` által meghatározott állomány létezik, visszatérési értéke 1, máskülönben 0; a keresési útvonal elsősorban a pillanatnyi könyvtár, továbbá a `+L` parancssori kapcsolóval megadott elérési utak.
- `str(A, L, P)`: az `A` lebegőpontos értéket szöveggé alakítja – az `L` határozza meg a szöveg legnagyobb hosszúságát, a `P` pedig a tizedesjegyek számát, ha a szám kisebb helyen is elfér, mint az `L` érték által megadott hosszúság, és a szöveget jobbra igazítja; a `P` értéke negatív szám is lehet, ekkor a függvény a bal oldalon

3. lista

```
//
// Filename: gomb_forog.pov
// Start rendering: povray anim.ini -i
// gomb_forog.pov
//
// anim.ini:
// Initial_Frame=0
// Final_Frame=360
// Initial_Clock=0
// Final_Clock=720
//
#include "colors.inc"
#include "stones.inc"

camera {
   location <3,4,5>
   look_at <0,0,0>
}

light_source {
       <1,5,1>
       1.2*White
}

#declare gomboc=sphere{
            <0,0,0>,1
            pigment {checker
                       pigment {Jade},
                       pigment {Black_Marble}
                    }
            }
object {
   gomboc
   rotate y*clock
}
```

4. lista

```
// `ltalÆnos  zenet
#debug " zenet a DEBUG sorba"
// VØgzetes hiba
#error "A szÆm tÆsi folyamat leÆll"
//  zenet a RENDER sorba
#render "Most szÆmolgatok..."
// Statisztikai adat
#statistics "A kØp elkØsz lt"
// Figyelmeztetøs
#warning "KØrem, ne kapcsolja ki
#a szÆm t gØpet, Øppen szÆmol"
```

keletkező üres karaktereket 0-val tölti fel.

- `strcmp(S1, S2)`: összehasonlítja a két szöveget; eredménye 0, ha az értékek tartalma megegyezik, pozitív szám, ha az ASCII-kódok szerinti rendezés során `S2` megelőzi `S1`-et, és negatív, ha `S1` előzi meg `S2`-t.
- `strlen(S1)`: az `S1`-ben található karakterek számával tér vissza és meghatározza a szöveg hosszúságát
- `strlwr(S1)`: a megadott szöveget kisbetűssé alakítja, tehát minden nagybetűt kisbetűre cserél ki, így az `strlwr(Szia Olvaso)` által visszaadott szöveg „szia olvaso” lesz.
- `substr(S1, P, L)`: az `S1` részét határozza meg a P karaktertől kezdve `L` számú karakter figyelembevételével, tehát a `substr(ALMA,1,2)` hívás a `LM`-részletet adja vissza, mert a PoV-Rayben a szöveg első karaktere a 0. indexet kapja; ha `P+L` nagyobb az `S1` karakterlánc hosszánál, a PoV-Ray hibát jelez.
- `strupr(S1)`: a megadott szöveget nagybetűssé alakítja.
- `val(S1)`: az `S1` szöveget számmá alakítja, vagyis visszatérési értéke lebegőpontos szám.

Ennyi lenne tehát azoknak a függvényeknek a tárháza, amelyeket animációkészítés során is használhatunk, viszont még nem tudjuk, miként lehetne őket a gyakorlatban is felhasználni. Az alábbiakban a vezérlőutasítások ismertetésével ehhez nyújtok segítséget.
Első csoportjuk a feltételek kezeléséhez szükséges úgynevezett feltételes utasítások, ide tartozik az `IF-ELSE`, az `IFDEF` és az `IFNDEF` utasítás. Az `IF-ELSE` utasítás formája a *1. listában* látható.
Itt a feltétel logikai kifejezés, és az ilyen kifejezés akkor igaz, ha az értéke különbözik 0-tól. Amennyiben egyenlő 0-val, a PoV-Ray értelmezése szerint hamis. Az alábbiakban látható példa alapján az utasítás használata talán könnyebben megérthető:

```
#if (file_exists("szentely.inc")
      #include "szentely.inc"
#else
      #declare szentely=sphere{<0,0,0>,1.2}
#end
```

Itt azt adtuk a program tudtára, hogy ha létezik a ***szentely.inc*** állomány, akkor azt használjuk, ha pedig nem létezik, akkor a szentélyt egy gömbbel helyettesítjük. Megjegyezném, a ***szentely.inc*** tartalma célszerűen így kezdődhetne:

```
#define szentely=mesh{...}
```

A következő vezérlőutasítás, amiről szót kell ejtenünk, az `#IFDEF` nevet kapta. Használata akkor javasolt, ha valamilyen változó létezéséhez szeretnénk kötni valamely más utasítás értelmezését. A fenti példához hűen a kép számításának menetét alakítsuk át egy kicsit:

```
#if (file_exists("szentely.inc"))
      #include "szentely.inc"
#end
```

A lista alapján láthatjuk, hogyha létezik a szentélyt tartalmazó állomány, ellátjuk anyaggal, de az előző példával ellentétben nem helyettesítjük gömbbel, tehát amennyiben a külső állományban nem találjuk, nem is fogjuk használni. Az `#IFDEF` utasítás általános formája a következő:

```
#ifdef (szentely)
      object {szentely
            texture {...}
            }
```

```
#end
```

Az `#else` utasítást az előző esetek egyikénél sem kötelező használni, de az `#end` mindkét tárgyalt esetben kötelező. Ebbe a csoportba tartozik a harmadik feltételes utasítás is, az `#ifndef`. Használata szinte azonos az `#ifdef` alkalmazásával, a különbség csak annyi, hogy a PoV-Ray a `#ifndef` és az `#end` közötti utasításokat akkor veszi figyelembe, ha a megadott AZONOSITO nem létezik.
Külön soroltam a `#SWITCH CASE` és a `#RANGE` utasításokat, bár ezekkel is feltételes elágazásokat hozhatunk létre, ám általuk nagyobb szabadságot kapunk. Az utasítások általános formája a *2. listán* látható.
A listán alkalmazott módon változatos feltételeket szabhatunk egy esemény bekövetkeztének (például az objektumok létrehozása, elforgatása és eltolása során), amelyet alapvetően az AZONOSITO értéke határoz meg. Amennyiben ez megegyezik valamely `#case`-ágban meghatározott értékkel, az abban az ágban megadott utasítások kerülnek végrehajtásra egészen a következő `#break` vagy `#end` utasításig. Ha a `#case`-ágak egyike sem került végrehatásra, a következő utasítások a `#range` feltételek által meghatározottak lehetnek. Az itt megadottak akkor kerülnek végrehajtásra, ha az AZONOSITO értéke a `#range` utasítás MIN és MAX értéke közé esik. Ez esetben is a végrehajtás a következő `#break` vagy `#end` utasításig folytatódik. Az `#else`-ágban meghatározottakra pedig akkor kerülhet sor, ha az előzők egyike sem teljesült.
A PoV-Rayben ciklusszervező utasítás használatára is lehetőségünk nyílik, azonban egyedül az elölteszlelő ciklust alkalmazhatjuk. A `#WHILE` utasítással több hasonló objektumot hozhatunk létre, és az animációkészítés folyamán a tárgyakon, a kamerákon és a fényforrásokon egyaránt átalakításokat hajthatunk végre. Szintén kifejezésként kell megadnunk őket, amelynek értéke logikai igaz vagy hamis lesz. A cikluson belüli utasítások akkor kerülnek végrehajtásra, ha a kifejezés értéke igaz. Az utasítás általános formája a következő:

```
#while (FELT TEL)
       Utas tÆsok
       ...
#end
```

Szemléltetésképpen készítsünk öt gömböt egymás mellé:

```
#declare Count=0
#while (Count<5)
       sphere {
              <0,0,0>, 1
              translate x*2*Count
              }
       #declare Count=Count+1
#end
```

A vezérlőutasítások ismeretében már csak egy morzsányi ismeret szükséges első animációnk elkészítéséhez. A képkockák kiszámítása során egy állandóan változó értékre van szükségünk, ugyanis ez alapján tudjuk kiszámítani a mozgásokat és az egyéb változó értékeket. A PoV-Rayben ez az érték a `clock` változó, amelynek kezdő és végső értékét, valamint növekményét a számolás megkezdése előtt be lehet állítani. A beállításokat egy kezdő értéket megadó állomány segítségével végezhetjük el, amely rendszerint ***.ini*** kiterjesztésű. Az animáció kiszámításához legalább két értéket meg kell határoznunk: az egyik az `Initial_frame`, a másik a `Final_Frame`. Ezekkel adjuk a PoV-Ray tudtára, hogy az animáció milyen hosszú lesz, a `clock` változó pedig alapértelmezésben 0-tól 1-ig változik, azonban az `Initial_clock`-ot és a `Final_clock`-ot az `.ini` értékeinek segítségével változtathatjuk meg. Az alábbi példánkkal a könnyebb megértést kívánjuk elősegíteni:

```
Initial_Frame=0
Finale_Frame=360
Initial_Clock=0
Final_Clock=720
```

Itt úgy határoztuk meg az animációhoz szükséges értékeket, hogy a teljes animáció 360 képkockából álljon és a `clock` változó értéke 0–720-ig változzon. Ezt felhasználhatjuk például arra, hogy egy gömböt kétszer teljesen körbeforgassunk a *3. listán* látható módon.
Az animáció kiszámítására a következő paraméterezéssel vehetjük rá a PoV-Rayt:

```
povray anim.ini -i gomb_forog.pov
```

vagyis meg kell adni, hogy melyik ***.ini*** állományt szeretnénk használni.
Természetesen ez csak egy egyszerű példa volt az animációra, hiszen a kamerát, a fényforrásokat és minden elképzelhető dolgot mozgathatunk, ezt azonban sorozatunk záró részében fogom bemutatni.
Ebben a részben csupán egyetlen kiegészítő adatot szeretnék még olvasóinkkal megosztani, ez a felhasználói üzenetek létrehozására vonatkozik. A PoV-Ray a különböző típusú üzenetek megjelenítésére különböző üzenetfolyamokat használ. Ezek mindegyike átirányítható vagy akár le is tiltható. Az alább felsoroltakat különböztetjük meg:

- `DEBUG`-üzeneteket, amelyek általában a fejlesztők számára hasznosak, bár a felhasználók is hasznosíthatják.
- A `FATAL`-üzenetsor olyan üzenetek megjelenítésére alkalmas, amelyek a PoV-Ray leállásához vezetnek.
- A `RENDER`-üzenetek arra használhatók, hogy a számítások során általános adatokat jelenítsünk meg.
- A `STATISTICS`-üzenetsor pedig minden képkocka kiszámítása után a számítások során készített kimutatásokat tartalmazza.
- A `WARNING`-üzenetek olyan hibákat jeleznek, amelyek bekövetkeztekor a számítási folyamat nem áll le, és ez az üzenet jelenik meg figyelmeztetésként.

A felsorolt öt üzenetsorba a felhasználók is írhatnak üzeneteket a *4. listán* látható utasítások segítségével.
Mindenkinek kellemes alkotást kívánok!

Fábián Zoltán
(dzooli@freemail.hu, dzooli@yahoo.com)
24 éves, jelenleg programozóként dolgozik. Szabadidejében szívesen kirándul, túrázik. Emellett szeret rajzolni, érdekli a 3D grafika és a Linuxszal kapcsolatban minden olyan program és programnyelv, amit még nem ismer vagy nem próbált ki.